# imreg_dft Documentation

***Release 2.0.1a***

**Christoph Gohlke, Matěj Týč**

**Jun 25, 2018**

# Contents


**1 General overview** **1**
- 1.1 Requirements . . . 1
- 1.2 Quickstart . . . 2
- 1.3 Notes . . . 2
- 1.4 References . . . 2

**2 Contents** **3**
- 2.1 Quickstart guide . . . 3
- 2.2 Command-line tool overview . . . 6
- 2.3 Advanced command-line . . . 9
- 2.4 Utility scripts . . . 12
- 2.5 Use as a Python module . . . 14
- 2.6 Conceptual-level documentation . . . 16
- 2.7 Developer reference . . . 18
- 2.8 User-centric changelog . . . 29

**3 Indices and tables** **31**

**Python Module Index** **32**

CHAPTER 1

# General overview

`imreg_dft` implements DFT*[0] -based technique for translation, rotation and scale-invariant image registration.

In plain language, `imreg_dft` implements means of calculating translation, rotation and scale variation between two images. It doesn't work with those images directly, but it works with their spectrum, using the log-polar transformation. The algorithm is described in[1] and possibly also in[2] and[3].

Fig. 1: The template (a), sample (b) and registered sample (c). This is the actual output of *sample in the cli section*

**Authors**

- Matěj Týč
- Christoph Gohlke

**Organization**

- Brno University of Technology, Brno, Czech Republic
- Laboratory for Fluorescence Dynamics, University of California, Irvine



## 1.1 Requirements

See the `requirements.txt` file in the project's root for the exact specification. Generally, you will need `numpy` and `scipy` for the core algorithm functionality.

Optionally, you may need:

[0] DFT stands for Discrete Fourier Transform. Usually the acronym FFT (Fast Fourier Transform) is used in this context, but this is incorrect. DFT is the name of the operation, whereas FFT is just one of possible algorithms that can be used to calculate it.

[1] An FFT-based technique for translation, rotation and scale-invariant image registration. BS Reddy, BN Chatterji. IEEE Transactions on Image Processing, 5, 1266-1271, 1996

[2] An IDL/ENVI implementation of the FFT-based algorithm for automatic image registration. H Xiea, N Hicksa, GR Kellera, H Huangb, V Kreinovich. Computers & Geosciences, 29, 1045-1055, 2003.

[3] Image Registration Using Adaptive Polar Transform. R Matungka, YF Zheng, RL Ewing. IEEE Transactions on Image Processing, 18(10), 2009.

- `pillow` for loading data from image files,
- `matplotlib` for displaying image output,
- `pyfftw` for better performance.

## 1.2 Quickstart

Head for the *corresponding section of the documentation*. Note that you can generate the documentation yourself!

1. Install the package by running `python setup.py install` in the project root.
2. Install packages that are required for the documentation to compile (see the `requirements_docs.txt` file.
3. Go to the `doc` directory and run `make html` there. The documentation should appear in the `_build` subfolder, so you may open `_build/html/index.html` with your web browser to see it.

## 1.3 Notes

The API and algorithms are quite good, but help is appreciated. `imreg_dft` uses semantic versioning, so backward compatibility of any kind will not break across versions with the same major version number.

`imreg_dft` is based on the code by Christoph Gohlke.

## 1.4 References

CHAPTER 2

Contents

## 2.1 Quickstart guide

### 2.1.1 Installation

Before installing `imreg_dft`, it is good to have *dependencies* sorted out. `numpy` and `scipy` should be installed using package managers on Linux, or downloaded from the web for OSX and Windows.

#### The easy(_install) way

You can get the package from PyPi, which means that if you have Python `setuptools` installed, you can install `imreg_dft` using `easy_install`. For a user (i.e. not system-wide) install, insert `--user` between `easy_install` and `imreg_dft`. User install does not require administrator priviledges, but you may need to add the installation directories to your system path, otherwise the *ird* script won't be visible.

```
[user@linuxbox ~]$ easy_install imreg_dft
```

If you have `pip` installed, you can use it instead of `easy_install`. `pip` even allows you to install from the source code repository:

```
[user@linuxbox ~]$ pip install git+https://github.com/matejak/imreg_dft.git
```

#### The source way (also easy)

The other means is to check out the repository and install it locally (or even run `imreg_dft` without installation). You will need the `git` version control system to obtain the source code:

```
[user@linuxbox ~]$ git clone https://github.com/matejak/imreg_dft.git
[user@linuxbox ~]$ cd imreg_dft
[user@linuxbox imreg_dft]$ python setup.py install
...
```

As with other Python packages, there is a `setup.py`. To install `imreg_dft`, run `python setup.py install`. Add the `--user` argument after `install` to perform user (i.e. not system-wide) install. As stated

in the previous paragraph, the user install does not require administrator priviledges, but you may need to add the installation directories to your system path, otherwise the *ird* script won't be visible.

If you want to try `imreg_dft` without installing it, feel free to do so. The package is in the `src` directory.

### 2.1.2 Quickstart

A succesful installation means that:

- The Python interpreter can import `imreg_dft`.
- There is the `ird` script available to you, e.g. running `ird --version` should not end by errors of any kind.

---

**Note:** If you have installed the package using `pip` or `easy_install`, you don't have the example files, images nor test files. To get them, download the source archive from PyPi or release archive from Github and unpack them.

---

#### Python examples

The following examples are located in the `resources/code` directory of the project repository *or its source tree* as `similarity.py` and `translation.py`. You can launch them from their location once you have installed `imreg_dft` to observe the output.

The full-blown similarity function that returns parameters (and the transormed image):

```
import os

import scipy as sp
import scipy.misc

import imreg_dft as ird


basedir = os.path.join('..', 'examples')
# the TEMPLATE
im0 = sp.misc.imread(os.path.join(basedir, "sample1.png"), True)
# the image to be transformed
im1 = sp.misc.imread(os.path.join(basedir, "sample3.png"), True)
result = ird.similarity(im0, im1, numiter=3)

assert "timg" in result
# Maybe we don't want to show plots all the time
if os.environ.get("IMSHOW", "yes") == "yes":
    import matplotlib.pyplot as plt
    ird.imshow(im0, im1, result['timg'])
    plt.show()
```

Or just the translation:

```
import os

import scipy as sp
import scipy.misc

import imreg_dft as ird


basedir = os.path.join('..', 'examples')
```

(continues on next page)

(continued from previous page)

```
# the TEMPLATE
im0 = sp.misc.imread(os.path.join(basedir, "sample1.png"), True)
# the image to be transformed
im1 = sp.misc.imread(os.path.join(basedir, "sample2.png"), True)
result = ird.translation(im0, im1)
tvec = result["tvec"].round(4)
# the Transformed IMaGe.
timg = ird.transform_img(im1, tvec=tvec)

# Maybe we don't want to show plots all the time
if os.environ.get("IMSHOW", "yes") == "yes":
    import matplotlib.pyplot as plt
    ird.imshow(im0, im1, timg)
    plt.show()

print("Translation is {}, success rate {:.4g}"
      .format(tuple(tvec), result["success"]))
```

#### Command-line script examples

Please see the *corresponding section* that is full of examples.

### 2.1.3 Do not forget

These steps should go before the *Quickstart* section, but they come now as this is a quickstart guide.

#### Tests

If you have *downloaded the source files*, you can run tests after installation. There are now unit tests and regression tests. You can execute them by going to the `tests` subdirectory and running

```
[user@linuxbox imreg_dft]$ cd tests
[user@linuxbox tests]$ make help
make[1]: Entering directory '/home/user/imreg_dft/tests'
Run either 'make check' or 'make check COVERAGE=<coverage command name>'
You may also append 'PYTHON=<python executable>' if you don't use coverage
make[1]: Leaving directory '/home/user/imreg_dft/tests'
[user@linuxbox tests]$ make check
...
```

If you have the `coverage` module installed, you also have a `coverage` (or perhaps `coverage2`) scripts in your path. You can declare that and therefore have the tests ran with coverage support:

```
[user@linuxbox tests]$ make check COVERAGE=coverage2
...
```

In any way, if you see something like

```
[user@linuxbox tests]$ make check
...
make[1]: Leaving directory '/home/user/imreg_dft/tests'
* * * * * * * * * * * * * * * * * * * * *
 Rejoice, tests have passed successfully!
* * * * * * * * * * * * * * * * * * * * *
```

it is a clear sign that there indeed was no error encountered during the tests at all.

### Documentation

Although you can read the documentation on readthedocs.org (bleeding-edge) and pythonhosted.org (with images), you can generate your own easily. You just have to check out the `requirements_docs.txt` file at the root of the project and make sure you have all modules that are mentioned there. You also need to have `imreg_dft` installed prior to documentation generation.

Also be sure to have *the source files*. In the source tree, go to the `doc` directory there and run `make html` or `make latexpdf` there.

## 2.2 Command-line tool overview

The package contains three Python CLI (command-line interface) scripts. Although you are more likely to use the `imreg_dft` functionality from your own Python programs, you can still have some use to the `ird` front-end.

There are these main reasons why you would want to use it:

- Quickly learn whether `imreg_dft` is relevant for your use case.
- Quickly tune the advanced image registration settings.
- Use `ird` in a script and process batches of images instead of one-by-one. (`ird` can't do it, but you can call it in the script, possibly using GNU Paralell.)

Additionally, there are two more scripts — see their documentation in the *utilities section*.

### 2.2.1 General considerations

Generally, you call `ird` with two images, the first one being the `template` and the second one simply the `subject`. If you are not sure about other program options, run it with `--help` or `--usage` arguments:

```
[user@linuxbox examples]$ ird -h
usage: ird [-h] [--lowpass LOW_THRESH,HIGH_THRESH]
           [--highpass LOW_THRESH,HIGH_THRESH] [--cut LOW_THRESH,HIGH_
↪THRESH]
           [--resample RESAMPLE] [--iters ITERS] [--extend PIXELS]
           [--order ORDER] [--filter-pcorr FILTER_PCORR] [--print-result]
           [--print-format PRINT_FORMAT] [--tile] [--version]
           [--angle MEAN[,STD]] [--scale MEAN[,STD]] [--tx MEAN[,STD]]
           [--ty MEAN[,STD]] [--output OUTPUT] [--loader {pil,mat,hdr}]
           [--loader-opts LOADER_OPTS] [--help-loader] [--show]
           template subject

...
```

The notation `[foo]` means that specifying content of brackets (in this case `foo`) is optional. For example, let's look at a part of help `[--angle MEAN[,STD]]`. The outer square bracket means that specifying `--angle` is optional. However, if you specify it, you have to include the mean value as an argument, i.e. `--angle 30`. The inner square brackets then point out that you may also specify a standard deviation, in which case and you separate it from the mean using comma: `--angle 30,1.5`. There are sanity checks present, so you will be notified if you commit a mistake.

So only the input arguments are obligatory. Typically though, you will want to add arguments to also get the result:

1. Take an instant look at the registration result — use the `--show` argument.
2. Learn the registration parameters: Use the `--print-result` argument (explained in greater detail below).
3. Save the transformed subject: Use the `--output` argument.

The image registration works best if you have images that have features in the middle and their background is mostly uniform. If you have a section of a large image and you want to use registration to identify it, *most likely, you will not succeed*.

For more exhaustive list of known limitation, see the section *Caveats*.

### 2.2.2 Quick reference

1. Quickly find out whether it works for you, having the results (optionally) shown in a pop-up window and printed out. We assume you stand in the root of `imreg_dft` cloned from the git repo (or *downloaded from the web*).

```
[user@linuxbox imreg_dft]$ ird resources/examples/sample1.png resources/
↪examples/sample2.png --show --print-result
scale: 1 +-0.003212
angle: 0 +-0.1125
shift (x, y): -19, 79 +-0.25
success: 1
```

   The output tells you what has to be done with the `subject` so it looks like the `template`.

   **Warning:** Keep in mind that images have the zero coordinate (i.e. the origin) in their upper left corner!

2. You can have the results print in a defined way. First of all though, let's move to the examples directory:

```
[user@linuxbox imreg_dft]$ cd resources/examples
[user@linuxbox examples]$ ird sample1.png sample2.png --print-result --print-
↪format 'translation:%(tx)d,%(ty)d\n'
translation:-19,79
```

   You can get an up-to-date listing of possible values you can print using the help argument. Generally, you can get back the values as well as confidence interval half-widths that have a `D` prefix. For example there is `angle` and `Dangle`; in case that the method doesn't fail misreably, the true angle will not differ from `angle` more than over `Dangle`.

3. Let's try something more tricky! The first and third examples are rotated against each other and also have a different scale.

```
[user@linuxbox examples]$ ird sample1.png sample3.png --print-result --show
scale: 1.2475 +-0.004006
angle: -30.0493 +-0.1125
shift (x, y): 34.7935, 72.159 +-0.25
success: 0.9034
```

4. And now something even more tricky - when a part of the subject is cut out. The difference between the fourth and third image is their mutual translation which also causes that the feature we are matching against is cut out from the fourth one.

   Generally, we have to address the this The `--extend` option here serves exactly this purpose. It extends the image by a given amount of pixels (on each side) and it tries to blur the cut-out image beyond its original border. Although the blurring might not look very impressive, it makes a huge difference for the image's spectrum which is used for the registration. So let's try:

```
[user@linuxbox examples]$ ird sample1.png sample4.png --extend 20 --show --
↪print-result --iter 4
scale: 1.2474 +-0.00373
angle: -30.0789 +-0.10227
shift (x, y): 158.79, 94.9163 +-0.25
success: 0.6112
```

As we can see, the result is correct.

Extension can occur on-demand when the scale change or rotation operations result in image size growth. However, whether this will occur or not is not obvious, so it is advisable to specify the argument manually. In this example (and possibly in the majority of other examples) specifying the option manually is not needed.

> **Warning:** If the image extension by blurring is very different from how the image really looks like, the image registration will fail. Don't use this option until you become sure that it improves the registration quality.

5. Buy what do we actually get on output? You may wonder what those numbers mean. The output tells you *what has to be done* with the `image` *so it looks like* the `template`. The scale and angle information is quite clear, but the translation depends on the center of scaling and the center of rotation...

   So the idea is as follows — let's assume you have an image, an `imreg_dft` print output and all you want is to perform the image transformation yourself. The output describes what operations to perform on the image so it is close to the template. All transformations are performed using scipy.ndimage.interpolate package and you need to do the following:

   (a) Call the `zoom` function with the provided scale. The center of the zoom is the center of the subject.

   (b) Then, rotate the subject using the `rotate` function, specifying the given angle. The center of the rotation is again the center of the subject.

   (c) Finally, translate the subject using the `shift` function. Remember that the `y` axis is the first one and `x` the second one.

   (d) That's it, the subject should now look like the template.

6. Speaking of which, you can have the output saved to a file. This is handy for example if you record the same thing with different means (e.g. a cell recorded with multiple microscopes) and you want to examine the difference between them on a pixel-by-pixel basis. In order to be able to exploit this feature to its limits, read about `loaders`, but you can simply try this example:

```
[user@linuxbox examples]$ ird sample1.png sample3c.jpg --iter 3 --print-result␣
↪--output color.jpg
scale: 1.2493 +-0.004012
angle: -30.0184 +-0.1125
shift (x, y): 34.9015, 72.786 +-0.25
success: 0.6842
```

   To sum it up, the registration is a process performed with images somehow converted to grayscale (for example as the average across all color chanels). However, as soon as the transformation is known, an RGB image can be transformed to match the template and saved in full color.

### 2.2.3 Loaders

`ird` can support a wide variety of input formats. It uses an abstract means of how to load and save an image.

To cut the long story short — you probably want to autodetection of how to load an image based on the file extension. The list of available loaders is obtained by passing the `--help-loader`. To inquire about meaning of individual options, also specify a loader on the same command-line, e.g. pass `--loader pil`.

To pass an option to change loader properties pass a `--loader-opts` argument. It accepts comma-separated `option name=value` pairs, so for example the `mat` loader understands `--loader-opts in=imgdata, out=timgdata`. Note that all loaders have access to passed options.

The loaders concept functionality is limited by now, but it can be extended easily by writing code. See the *developer documentation* to learn the background. If you miss some functionality, you are kindly invited to create a pull request!

### 2.2.4 Caveats

There are known weak points of `ird`. Although they are explained in other places in the text, we sum them up here:

**Extending images.** Due to the fact that the spatial frequencies spectrum is used, the border of images are become important. We address it here by extending the image, but it often doesn't work well.

**Sub-pixel resolution.** This is a tricky matter. Since the frequency spectrum is used, neither linear or cubic interpolation guarantee an improvement. However, the log-polar transform is used with linear interpolation, since it has been observed that it has a positive impact on the result. For a more correct approach, you can try the resampling feature, but beware — you have to have correctly sampled (i.e. not undersampled) input for it to work.

**Big template.** If the template presents a wider field of view than the image, you may or may not be successful when using the `--tile` option. The current implementation is flaky.

**Success value.** The `Success` that is reported has an unclear meaning. And its behavior is also quite dodgy.

## 2.3 Advanced command-line

Apart from simple problems, the command-line is able to solve more tricky cases such as cases when

- one of the images has its spectrum cut (e.g. when different objectives were used),
- you want to apply interpolation (you have nice images and desire sub-pixel precision),
- one (or more) of rotation, scaling, or translation is known (so you want to help the process by narrowing possible values down) and
- the subject's field of view is a (small) subset of the template's (which is a huge obstacle when not taken care of in any way).

### 2.3.1 Frequency spectrum filtering

If you want to even images spectra, you want to use low-pass filters. This happens for example if you acquire photos of sample under a microscope using objective lenses with different numerical aperture. The fact that spectra don't match in high frequencies may confuse the method.

On the contrary, if you have some low-frequency noise (such as working with inverted images), you need a high-pass filter.

**`--lowpass`, `--highpass`** These two concern filtration of images prior to the registration. There can be multiple reasons why to filter images:

- One of them is filtered already due to conditions beyond your control, so by filtering them again just brings the other one on the par with the first one. As a side note, filtering in this case should make little to no difference.
- A part of spectrum contains noise which you want to remove.
- You want to filter out low frequencies since they are of no good when registering images anyway.

The filtering works like this:

The domain of the spectrum is a set of spatial frequencies. Each spatial frequency in an image is a vector with a $x$ and $y$ components. We norm the frequencies by stating that the highest value of a compnent is 1. Next, define the *value* of spatial frequency as the (euclidean) length of the normed vector. Therefore the spatial frequencies of greatest values ($\sqrt{2}$) are (1, 1), (1, -1) etc.

An argument to a `--lowpass` or `--highpass` option is a tuple composed of numbers between 0 and 1. Those relate to the value of spatial frequencies it affects. For example, passing `--lowpass 0.2,0.3` means that spatial frequencies with value ranging from 0 to 0.2 will pass and those with value higher than 0.4 won't. Spatial frequencies with values in-between will be progressively attenuated.

Therefore, the filter value $f(x)$ based on spatial frequency value $x$ is

$$f(x) = \begin{cases} 1 & : x \leq 0.2 \\ -10\,x + 3 & : 0.2 < x \leq 0.3 \\ 0 & : 0.3 < x \end{cases}$$

where the middle transition term is a simplified form of $(0.3 - x)/(0.3 - 0.2)$.

**Note:** A continuous high-pass filtration is already applied to the image. The filter is $(1 - \cos[\pi\ x/2])^2$

You can also filte the phase correlation process itself. During the registration, some false positives may appear. This may occur for example if the image pattern is not sampled very densely (i.e. close or even below the Nyquist frequency), ripples may appear near edges in the image.

These ripples basically interfere with the algorithm and the phase correlation filtration may overcome this problem. If you apply the following filter, only the convincing peaks will prevail.

**`--filter-pcorr`** The value you supply to this filter is radius of minimum filter applied to the cross power spectrum. Typically 2–5 will accomplish the goal. Higher values are not recommended, but see for yourself.

### 2.3.2 Interpolation

You can try to go for sub-pixel precision if you request resampling of the input prior to the registration. Resampling can be regarded as an interpolation method that is the only correct one in the case when the data are sampled correctly. As opposed to well-known 2D interpolation methods such as bilinear or bicubic, resampling uses the $sinc(x) = sin(x)/x$ function, but it is usually implemented by taking a discrete Fourier transform of the input, padding the spectrum with zeros and then performing an inverse transform. If you try it, results are not so great:

```
[user@linuxbox examples]$ ird sample1.png sample3.png --resample 3 --iter 4 --
↪print-result
scale: 1.2492 +-0.001619
angle: -30.0507 +-0.0375
shift (x, y): 104.977, 218.134 +-0.08333
success: 0.2343
```

However, resampling can result in artifacts near the image edges. This is a known phenomenon that occurs when you have an unbounded signal (i.e. signal that goes beyond the field of view) and you manipulate its spectrum. Extending the image and applying a mild low-pass filter can improve things considerably.

The first operation removes the edge artifact problem by making the opposing edges the same and making the image seamless. This removes spurious spatial frequencies that appear as a + pattern in the image's power spectrum. The second one then ensures that the power spectrum is mostly smooth after the zero-pading, which is also good.

```
[user@linuxbox examples]$ ird sample1.png sample3.png --resample 3 --iter 4 --
↪lowpass 0.9,1.1 --extend 10 --print-result
scale: 1.249 +-0.001557
angle: -30.0446 +-0.035714
shift (x, y): 104.953, 218.093 +-0.08333
success: 0.2207
```

As we can see, both the scale and angle were determined extremely precisely. So, a warning for those who skip the ordinary text:

**`--resample`** The option accepts a (float) number specifying the resampling coefficient, so passing 2.0 means that the images will be resampled so its dimension become twice as big.

**Warning:** The `--resample` option offers the potential of sub-pixel resolution. However, when using it, be sure to start off with (let's say) `--extend 10` and `--lowpass 0.9,1.1` to exploit it. Then, experiment with the settings until the results look best.

### 2.3.3 Using constraints

`imreg_dft` allows you to specify a constraint on any transformation. It works the same way for all values. You can specify the expected value and the uncertainity by specifying a mean ($\mu$) and standard deviation ($\sigma$) of the variable.

Values is proportionally reduced in the phase correlation phase of the algorithm. Here is what happens if we force a wrong angle:

When the template and subject are the same, the algorithm would have no problems with the registration. If we force a certain angle by specifying a value with a low $\sigma$, the result is obeyed. However, the algorithm is actually quite puzzled and it would fail if we didn't specify the scale constraint.

```
[user@linuxbox examples]$ cd constraints
[user@linuxbox constraints]$ ird tricky.png tricky.png --angle 170,1 --scale 1,0.
↪05 --print-result
scale: 0.9934 +-0.001803
angle: 169.836 +-0.057618
shift (x, y): -17.3161, 23.2186 +-0.25
success: 0.2317
```

When we place a less restrictive constraint, a locally optimal registration different from the mean (180° vs 170°) is found:

Fig. 1: The template and the subject at the same time (a), registered with `--angle 170,10` (b) and registered with `--angle 170,1` (c).

You can use (separately or all at once) options `--angle`, `--scale`, `--tx` and `--ty`. Notice that since the translation comes after scale change and rotation, it doesn't make much sense to use either `--tx` or `--ty` without having strong control over `--angle` *and* `--scale`.

You can either:

- Ignore the options — the default are null constraints.
- Specify a null constraint explicitly by writing the delimiting comma not followed by anything (i.e. `--angle -30,`).
- Pass the mean argument but omit the stdev part, in which case it is assumed zero (i.e. `--angle -30` is the same as `--angle -30,0`). Zero standard deviation is directive — the angle value that is closest to -30 will be picked.
- Pass both parts of the argument — mean and stdev, i.e. `--angle -30,1`, in which case angles below -33 and above -27 are virtually ruled out.

---

**Note:** The Python argument parsing may not like argument value `-30,2` because `-` is the single-letter argument prefix and `-30,2` is not a number (unlike `-30.2`). On unix-like systems, you may circumvent this by writing `--angle ' -30,2'`. Now, the argument value begins by space (and not by the dash) which doesn't make any trouble further down the road.

---

### 2.3.4 Large templates

`imreg_dft` works well on images that show the same object that is contained within the field of view with an uniform background. However, the method fails when the fields of view don't match and are in subset-superset relationship.

Normally, the image will be "extended", but that may not work. Therefore, if the subject is the *smaller* of the two, i.e. `template` encompasses it, you can use the `--tile` argument. Then, the template will be internally subdivided into tiles (of similar size to the subject's size) and individual tiles will be matched against the subject and the tile that matches the best will determine the transformation.

The `--show` option will show matching over the best fitting tile and you can use the `--output` option to save the registered subject (that is enlarged to the shape of the template).

### 2.3.5 Result

The following result demonstrates usage of `ird` under hard conditions. .. There are two images, the template is taken from confocal microscope (a), the subject is a phase acquired using a digital holographic microscope[4].

Pretty much everything that could go wrong indeed went:

- Spectrums were not matching (the template is sharper than the subject).
- The template obviously shows a wider area than the subject.
- The images actually differ in many aspects.

Well, at least the scale and angle are somehow known, so it is possible to use constraints in a mild way.

The question is — will it register?

And the answer obviously is — yes, if you use right options.

One of the right commands is

```
[user@linuxbox examples]$ cd tiling
[user@linuxbox tiling]$ ird big.png small.png --tile --print-result
scale: 0.51837 +-0.004024
angle: 40.5952 +-0.16854
shift (x, y): 156.397, 136.526 +-0.25
success: 0.6728
```

Fig. 2: The template (a), subject (b) and registered subject (c). Try for yourself with the `--show` argument and be impressed!

## 2.4 Utility scripts

There are two scripts that complement the main *ird script*. Those are:

- Transfomation tool — good if you know relation between the template and subject and you just want to transform the subject in the same way as the `ird` tool.
- Incpection tool — intended for gaining insight into the phase correlation as such. Especially handy in cases when something goes wrong or when you want to gain insight into the phase correlation process.

### 2.4.1 Transormation tool

The classical use case of phase correlation is a situation when you have the subject, the template, and your goal is to transform the subject in a way that it matches the template. The transformation parameters are unknown, and the purpose of phase correlation is to compute them.

So the use case consists of two sub-cases:

- Compute relation of the two images, and
- transform the subject according to the result of the former.

[4] Coherence-controlled holographic microscope. Pavel Kolman and Radim Chmelík, Opt. Express 18, 21990-22003 (2010) http://www.opticsinfobase.org/vjbo/abstract.cfm?URI=oe-18-21-21990

The `imreg_dft` project enables you to do all using the `ird` script, but those two steps can be split — the first can be done by `ird`, whereas the second by `ird-tform`. The transform parameters can be specified as an argument, or they can be read from stdin.

Therefore, those two one-liners are equivalent — the file `subject-tformed.png` is the rotated subject `sample3.png`, so it matches the template `sample1.png`. Also note that the `ird` script alone will do the job faster.

```
[user@linuxbox examples]$ ird sample1.png sample3.png -o subject-tformed.png
[user@linuxbox examples]$ ird sample1.png sample3.png --print-result | ird-tform␣
↪sample3.png --template sample1.png subject-tformed.png
```

Technically, the output of `ird` alone should be identical to `ird-tform`.

### 2.4.2 Inspection tool

The phase correlation method is built around the Fourier transform and some relatively simple concepts around it.

Although the phase correlation is an image registration technique that is highly regarded by field experts, it may produce unwanted results. In order to find out what is going on, you can request visualizations of various phases of the registration process.

Typically, you choose the template, the subject and instead of performing casual phase correlation using `ird`, you use `ird-show`. Then, the normal phase correlation takes place and various stages of it are output in form of images (see the `--display` argument of `ird-show`).

{% if READTHEDOCS %} .. warning:

```
Some images are not available on readthedocs.
To see them, refer to the `uploaded documentation <>`_.
We apologize for the inconvenience which is related to `this readthedocs␣
↪limitation <https://github.com/rtfd/readthedocs.org/issues/1054>`_.
```

{% endif %}

Fig. 3: Filtered `sample1.png` and `sample3n.jpg` respectively.

For example, consider the display of the final part of phase correlation — the translation (between the `sample1.png` and `sample3n.jpg` from the examples):

Fig. 4: The left part shows the cross-power spectrum (CPS) of the template and rotated and scaled subject as-is. The right part shows the CPS of the template and rotated and scaled subject as-is, where the rotation angle is increased by 180°.

As we can see, the success value is much higher for the first figure, so unless there is a angle constraint, the registration procedure will assume that the match in the first figure corresponds to the successful final step of the image registration. You can visualize any subset from the table below:

| code | filename stem | what it is |
|---|---|---|
| `i` | `ims_filt` | supplied template and subject after application of common filters |
| `s` | `dfts_filt` | log-abs of frequency spectra of supplied template and subject (after application of common filters) |
| `l` | `logpolars` | log-polar transform of the former |
| `1` | `sa` | insight into scale-angle phase correlation |
| `a` | `after_tform` | after application of the first part of phase correlation — the angle-scale transform |
| `2` | `t_0`, `t_180` or `t` if terse | insight into translation phase correlation |
| `t` | `tiles_successes`, `tiles_decomp` | insight into the *tiling functionality* |

## 2.5 Use as a Python module

The following is the public API of `imreg_dft`.

See the *Python examples* section to see how to use it.

### 2.5.1 imreg module

FFT based image registration. — main functions

`imreg_dft.imreg.`**`imshow`**(*im0*, *im1*, *im2*, *cmap=None*, *fig=None*, ***kwargs*)

Plot images using matplotlib. Opens a new figure with four subplots:

```
+----------------------+---------------------+
|                      |                     |
|   <template image>   |   <subject image>   |
|                      |                     |
+----------------------+---------------------+
| <difference between  |                     |
|  template and the    |<transformed subject>|
| transformed subject> |                     |
+----------------------+---------------------+
```

**Parameters**

- **im0** (*np.ndarray*) – The template image
- **im1** (*np.ndarray*) – The subject image
- **im2** – The transformed subject — it is supposed to match the template
- **cmap** (*optional*) – colormap
- **fig** (*optional*) – The figure you would like to have this plotted on

**Returns** The figure with subplots

**Return type** matplotlib figure

`imreg_dft.imreg.`**`similarity`**(*im0*, *im1*, *numiter=1*, *order=3*, *constraints=None*, *filter_pcorr=0*, *exponent='inf'*, *reports=None*)

Return similarity transformed image im1 and transformation parameters. Transformation parameters are: isotropic scale factor, rotation angle (in degrees), and translation vector.

A similarity transformation is an affine transformation with isotropic scale and without shear.

**Parameters**

- **im0** (*2D numpy array*) – The first (template) image
- **im1** (*2D numpy array*) – The second (subject) image
- **numiter** (*int*) – How many times to iterate when determining scale and rotation
- **order** (*int*) – Order of approximation (when doing transformations). 1 = linear, 3 = cubic etc.
- **filter_pcorr** (*int*) – Radius of a spectrum filter for translation detection
- **exponent** (*float or 'inf'*) – The exponent value used during processing. Refer to the docs for a thorough explanation. Generally, pass "inf" when feeling conservative. Otherwise, experiment, values below 5 are not even supposed to work.
- **constraints** (*dict or None*) – Specify preference of seeked values. Pass None (default) for no constraints, otherwise pass a dict with keys `angle`, `scale`, `tx` and/or `ty` (i.e. you can pass all, some of them or none of them, all is fine). The value of a key is supposed to be a mutable 2-tuple (e.g. a list), where the first value is related to

the constraint center and the second one to softness of the constraint (the higher is the number, the more soft a constraint is).

More specifically, constraints may be regarded as weights in form of a shifted Gaussian curve. However, for precise meaning of keys and values, see the documentation section *Using constraints*. Names of dictionary keys map to names of command-line arguments.

**Returns** Contains following keys: `scale`, `angle`, `tvec` (Y, X), `success` and `timg` (the transformed subject image)

**Return type** dict

---

**Note:** There are limitations

- Scale change must be less than 2.
- No subpixel precision (but you can use *resampling* to get around this).

---

imreg_dft.imreg.**similarity_matrix**(*scale*, *angle*, *vector*)

Return homogeneous transformation matrix from similarity parameters.

Transformation parameters are: isotropic scale factor, rotation angle (in degrees), and translation vector (of size 2).

The order of transformations is: scale, rotate, translate.

imreg_dft.imreg.**transform_img**(*img*, *scale=1.0*, *angle=0.0*, *tvec=(0, 0)*, *mode='constant'*, *bgval=None*, *order=1*)

Return translation vector to register images.

**Parameters**

- **img** (*2D or 3D numpy array*) – What will be transformed. If a 3D array is passed, it is treated in a manner in which RGB images are supposed to be handled - i.e. assume that coordinates are (Y, X, channels). Complex images are handled in a way that treats separately the real and imaginary parts.
- **scale** (*float*) – The scale factor (scale > 1.0 means zooming in)
- **angle** (*float*) – Degrees of rotation (clock-wise)
- **tvec** (*2-tuple*) – Pixel translation vector, Y and X component.
- **mode** (*string*) – The transformation mode (refer to e.g. `scipy.ndimage.shift()` and its kwarg `mode`).
- **bgval** (*float*) – Shade of the background (filling during transformations) If None is passed, *imreg_dft.utils.get_borderval()* with radius of 5 is used to get it.
- **order** (*int*) – Order of approximation (when doing transformations). 1 = linear, 3 = cubic etc. Linear works surprisingly well.

**Returns** The transformed img, may have another i.e. (bigger) shape than the source.

**Return type** np.ndarray

imreg_dft.imreg.**transform_img_dict**(*img*, *tdict*, *bgval=None*, *order=1*, *invert=False*)

Wrapper of *transform_img()*, works well with the *similarity()* output.

**Parameters**

- **img** –
- **tdict** (*dictionary*) – Transformation dictionary — supposed to contain keys "scale", "angle" and "tvec"
- **bgval** –
- **order** –

- **invert** (*bool*) – Whether to perform inverse transformation — doesn't work very well with the translation.

**Returns**

**See also:**

*transform_img()*

**Return type** np.ndarray

imreg_dft.imreg.**translation**(*im0*, *im1*, *filter_pcorr=0*, *odds=1*, *constraints=None*, *reports=None*)

Return translation vector to register images. It tells how to translate the im1 to get im0.

**Parameters**

- **im0** (*2D numpy array*) – The first (template) image
- **im1** (*2D numpy array*) – The second (subject) image
- **filter_pcorr** (*int*) – Radius of the minimum spectrum filter for translation detection, use the filter when detection fails. Values > 3 are likely not useful.
- **constraints** (*dict or None*) – Specify preference of seeked values. For more detailed documentation, refer to *similarity()*. The only difference is that here, only keys `tx` and/or `ty` (i.e. both or any of them or none of them) are used.
- **odds** (*float*) – The greater the odds are, the higher is the preferrence of the angle + 180 over the original angle. Odds of -1 are the same as inifinity. The value 1 is neutral, the converse of 2 is 1 / 2 etc.

**Returns**

**Contains following keys: angle, tvec (Y, X),** and `success`.

**Return type** dict

## 2.6 Conceptual-level documentation

### 2.6.1 Image registration procedure

Now, let's examine the *imreg_dft.imreg.similarity()* function. It estimates the scale, rotation and translation relationship between the images and we will take a look what are the means by which the end-result is obtained.

The image registration is carried out as follows:

1. Images (template and subject) are loaded in a form of 2D or 3D numpy arrays, where coordinates have this meaning: `(y, x [, channel])`.
2. Both images are filtered. Typically, low spatial frequencies are stripped, because they are not useful for the phase correlation. This is done in *imreg_dft.utils.imfilter()* and also later with the help of *imreg_dft.imreg._logpolar_filter()*.
3. If requested, both images are resampled (in other words, upscaled, see *imreg_dft.tiles.resample()*).
4. If necessary, both images are extended so that their shapes match. Implementation in *imreg_dft.utils.embed_to()*, gets called by *imreg_dft.tiles._preprocess_extend()*,
5. Phase correlation is performed to determine angle–scale change (*imreg_dft.imreg.similarity()*, *imreg_dft.imreg._get_ang_scale()*):
   (a) Images are apodized (so they are seamless with respect of their borders) in *imreg_dft.imreg._get_ang_scale()* by calling *imreg_dft.utils._apodize()*.

   (b) Amplitude of the Fourier spectrum is calculated and the log-polar transformation is performed (*imreg_dft.imreg._logpolar()*).

   (c) Phase correlation is performed on that log-polar spectrum amplitude (*imreg_dft.imreg._phase_correlation()*).

   (d) Source image is transformed to match the template (according to the angle and scale change — *imreg_dft.imreg.transform_img()*).

6. Second round of phase correlation is performed to determine translation (*imreg_dft.imreg.translation()*). Images are already somewhat apodized and compatible (this is ensured in the previous step).

   (a) Phase correlation on spectra of the template and the transformed subject is performed.

   (b) Phase correlation on spectra of the template and the transformed subject rotated over 180° is performed.).

   (c) Results of both operations are compared and the one that is more successful serves as final determination of angle and true translation vector. This is due to the fact that the determination of angle using phase correlation is ambiguous — an angle and the angle + 180° satisfy it in the same fashion.

7. The result (transformation parameters, transformed subject, ...) is saved to a dictionary.

8. If a transformed subject is requested (e.g. if you want to compare it with the template pixel-by-pixel), it is made (by undoing extending and resampling operations — *imreg_dft.utils.unextend_by()*, *imreg_dft.tiles._postprocess_unextend()*).

### Translation

The phase correlation method is able to guess translation from the phase of image's spectrum (i.e. its Fourier transform). For more in-depth reading consult the Wikipedia entry. The short-hand explanation is that translation of function is possible by taking its spectrum, multiplying it by a complex function and inverting it back to image. Hence, when we have two shifted images, it is obviously possible to guess their translation from their spectra.

The image is an array of real numbers, therefore its spectrum is symmetric in a way. This is the reason why the translation is checked first of all on the two images, and then one of them is rotated 180 degrees and the check is repeated.

Performing phase correlation on the two images means:

- Spectra are calculated from respective images.
- Cross-power spectrum is calculated:

$$R = \frac{F_1 \bar{F}_2}{|F_1||F_2| + \varepsilon}$$

  where $F_{1,2}$ are Fourier transforms (i.e. spectra) of input images ($\bar{F}_2$ is a complex conjugate of $F_2$) and $\varepsilon$ is a very small positive real number. Note that it is normalized, so $\max R = 1$ (when not taking $\varepsilon$ into account).

- The input for phase correlation is calculated:

$$R_i = |F^{-1}(R)|,$$

  where $R$ is the cross-power spectrum and $F^{-1}$ is the inverse Fourier transform operator.

- The shifted cross-power spectrum is passed to *imreg_dft.utils.argmax_translation()* and translation vector and success value are returned.

  There are arguments passed to the translation estimate function:

  – `filter_pcorr`: Radius of a minimum filter. Typically, when images are just translated, a translation one pixel off is still quite good. The phase correlation method heavily relies on image's high frequencies and sometimes there may be one image translation that looks good from the phase correlation perspective. If we apply a minimum filter, those false positives disappear, whereas the true result is affected much less.

  - `constraints`: Sometimes, we roughly know how the translation should be. Therefore, we can specify it, and it will be less likely that it will pick solution that is more favorable, but differs from the constraint.
  - `report`: When something goes wrong, it is good to have some insight into how internal data inside of the function looked like.
- The function outputs the translation vector and a success value — the value of ... (to be continued)

#### Rotation and scale

### 2.6.2 The front-end

## 2.7 Developer reference

### 2.7.1 Modules

The following is a functionality that may be useful, but it is not considered as public API and it may somehow evolve over time.

#### loader module

The module contains a layer of functionality that allows abstract saving and loading of files.

A loader class inherits from *Loader*. A singleton class `LoaderSet` is the main public interface of this module. available as a global variable `LOADERS`. It keeps track of all registered loaders and takes care after them (presents them with options, requests etc.) Loaders are registered as classes using the decorator `loader()`.

The concept is that there is one loader instance per file loaded. When we want to save a file, we use a loading loader to provide data to save and then we instantiate a saving loader (if needed) and save the data.

Individual loaders absolutely have to implement methods *Loader._save()* and *Loader._load2reg()*.

This module facilitates integration of its functionality by defining `update_parser()` and *settle_loaders()*. While the first one can add capabilities to a parser (or parser group), the second one updates `LOADERS` accordingly while given parsed arguments.

Rough edges (but not rough enough to be worth the trouble):

- You can't force different loaders for image, template and output. If you need this, you have to rely on autodetection based on file extension.
- Similarly, there is a problem with loader options — they are shared among all loaders. This is both a bug and a feature though.
- To show the loaders help, you have to satisfy the parser by specifying a template and image file strings (they don't have to be real filenames tho).

**class** imreg_dft.loader.**Loader**

**_save**(*fname*, *tformed*)
To be implemented by derived class. Save data to fname, possibly taking into account previous loads and/or options passed upon the class creation.

**_load2reg**(*fname*)
To be implemented by derived class. Load data from fname in a way that they can be used in the registration process (so it is a 2D array). Possibly take into account options passed upon the class creation.

**guessCanLoad**(*fname*)
Guess whether we can load a filename just according to the name (extension)

**load2reg**(*fname*)
Given a filename, it loads it and returns in a form suitable for registration (i.e. float, flattened, ...).

**save**(*fname*, *what*, *loader*)
Given the registration result, save the transformed input.

**spawn**()
Makes a new instance of the object's class BUT it conserves vital data.

imreg_dft.loader.**flatten**(*image*, *char*)
Given a layered image (typically (y, x, RGB)), return a plain 2D image (y, x) according to a spec.

**Parameters**

- **image** (*np.ndarray*) – The image to flatten
- **char** (*char*) – One of (R, G, B, or V (=value))

**Returns** np.ndarray - The 2D image.

imreg_dft.loader.**loader_of**(*lname*, *priority*)
A decorator interconnecting an abstract loader with the rest of imreg_dft It sets the "nickname" of the loader and its priority during autodetection

imreg_dft.loader.**settle_loaders**(*args*, *fnames=None*)
The function to be called as soon as args are parsed. It:

1. If requested by passed args, it prints loaders help and then exits the app
2. If filenames are supplied, it returns list of respective loaders.

**Parameters**

- **args** (*namespace*) – The output of argparse.parse_args()
- **fnames** (*list, optional*) – List of filenames to load

**Returns** list - list of loaders to load respective fnames.

### utils module

This module contains various support functions closely related to image registration. They are used mainly by the ird tool.

FFT based image registration. — utility functions

imreg_dft.utils.**_ang2complex**(*angles*)
Transform angle in degrees to complex phasor

imreg_dft.utils.**_apodize**(*what*, *aporad=None*, *ratio=None*)
Given an image, it apodizes it (so it becomes quasi-seamless). When ratio is None, color near the edges will converge to the same colour, whereas when ratio is a float number, a blurred original image will serve as background.

**Parameters**

- **what** – The original image
- **aporad** (*int*) – Radius [px], width of the band near the edges that will get modified
- **ratio** (*float or None*) – When None, the apodization background will be a flat color. When a float number, the background will be the image itself convolved with Gaussian kernel of sigma (aporad / ratio).

**Returns** The apodized image

imreg_dft.utils.**_argmax2D**(*array*, *reports=None*)
Simple 2D argmax function with simple sharpness indication

imreg_dft.utils.**_argmax_ext**(*array*, *exponent*)
Calculate coordinates of the COM (center of mass) of the provided array.

**Parameters**

- **array** (*ndarray*) – The array to be examined.
- **exponent** (*float or 'inf'*) – The exponent we power the array with. If the value 'inf' is given, the coordinage of the array maximum is taken.

**Returns** The COM coordinate tuple, float values are allowed!

**Return type** np.ndarray

imreg_dft.utils.**_calc_tform**(*shape*, *orig*, *scale*, *angle*, *tvec*, *newshape=None*)
probably not used

imreg_dft.utils.**_calc_tform_complete**(*shape*, *scale*, *angle*, *tvec*, *newshape=None*)

imreg_dft.utils.**_compensate_fftshift**(*vec*, *shape*)

imreg_dft.utils.**_complex2ang**(*cplx*)
Inversion of *_ang2complex()*

imreg_dft.utils.**_extend_array**(*arr*, *point*, *radius*)

imreg_dft.utils.**_getCut**(*big*, *small*, *offset*)
Given a big array length and small array length and an offset, output a list of starts of small arrays, so that they cover the big one and their offset is <= the required offset.

**Parameters**

- **big** (*int*) – The source length array
- **small** (*float*) – The small length

**Returns** list - list of possible start locations

imreg_dft.utils.**_get_angles**(*shape*)
In the log-polar spectrum, the (first) coord corresponds to an angle. This function returns a mapping of (the two) coordinates to the respective angle.

imreg_dft.utils.**_get_constraint_mask**(*shape*, *log_base*, *constraints=None*)
Prepare mask to apply to constraints to a cross-power spectrum.

imreg_dft.utils.**_get_dst1**(*pt*, *pts*)
Given a point in 2D and vector of points, return vector of distances according to Manhattan metrics

imreg_dft.utils.**_get_emslices**(*shape1*, *shape2*)
Common code used by *embed_to()* and *undo_embed()*

imreg_dft.utils.**_get_lograd**(*shape*, *log_base*)
In the log-polar spectrum, the (second) coord corresponds to an angle. This function returns a mapping of (the two) coordinates to the respective scale.

**Returns**

**2D np.ndarray of shape shape, -1 coord contains scales** from 0 to log_base ** (shape[1] - 1)

imreg_dft.utils.**_get_subarr**(*array*, *center*, *rad*)

**Parameters**

- **array** (*ndarray*) – The array to search
- **center** (*2-tuple*) – The point in the array to search around
- **rad** (*int*) – Search radius, no radius (i.e. get the single point) implies rad == 0

imreg_dft.utils.**_get_success**(*array*, *coord*, *radius=2*)
Given a coord, examine the array around it and return a number signifying how good is the "match".

**Parameters**

- **radius** – Get the success as a sum of neighbor of coord of this radius
- **coord** – Coordinates of the maximum. Float numbers are allowed (and converted to int inside)

**Returns** Success as float between 0 and 1 (can get slightly higher than 1). The meaning of the number is loose, but the higher the better.

imreg_dft.utils.**_highpass**(*dft*, *lo*, *hi*)

imreg_dft.utils.**_interpolate**(*array*, *rough*, *rad=2*)
Returns index that is in the array after being rounded.

The result index tuple is in each of its components between zero and the array's shape.

imreg_dft.utils.**_lowpass**(*dft*, *lo*, *hi*)

imreg_dft.utils.**_xpass**(*shape*, *lo*, *hi*)
Compute a pass-filter mask with values ranging from 0 to 1.0 The mask is low-pass, application has to be handled by a calling funcion.

imreg_dft.utils.**argmax_angscale**(*array*, *log_base*, *exponent*, *constraints=None*, *reports=None*)
Given a power spectrum, we choose the best fit.

The power spectrum is treated with constraint masks and then passed to *_argmax_ext()*.

imreg_dft.utils.**argmax_translation**(*array*, *filter_pcorr*, *constraints=None*, *reports=None*)

imreg_dft.utils.**decompose**(*what*, *outshp*, *coef*)
Given an array and a shape, it creates a decomposition of the array in form of subarrays and their respective position

**Parameters**

- **what** (*np.ndarray*) – The array to be decomposed
- **outshp** (*tuple-like*) – The shape of decompositions

**Returns** Decomposition — a list of tuples (subarray (np.ndarray), coordinate (np.ndarray))

**Return type** list

imreg_dft.utils.**embed_to**(*where*, *what*)
Given a source and destination arrays, put the source into the destination so it is centered and perform all necessary operations (cropping or aligning)

**Parameters**

- **where** – The destination array (also modified inplace)
- **what** – The source array

**Returns** The destination array

imreg_dft.utils.**extend_by**(*what*, *dst*)
Given a source array, extend it by given number of pixels and try to make the extension smooth (not altering the original array).

imreg_dft.utils.**extend_to**(*what*, *newdim*)
Given an image, it puts it in a (typically larger) array. To prevent rough edges from appearing, the containing array has a color that is close to the image's border color, and image edges smoothly blend into the background.

**Parameters**

- **what** (*ndarray*) – What to extend
- **newdim** (*tuple*) – The resulting dimension

imreg_dft.utils.**extend_to_3D**(*what*, *newdim_2D*)
Extend 2D and 3D arrays (when being supplied with their x–y shape).

imreg_dft.utils.**frame_img**(*img*, *mask*, *dst*, *apofield=None*)
Given an array, a mask (floats between 0 and 1), and a distance, alter the area where the mask is low (and roughly within dst from the edge) so it blends well with the area where the mask is high. The purpose of this is removal of spurious frequencies in the image's Fourier spectrum.

**Parameters**

- **img** (*np.array*) – What we want to alter
- **maski** (*np.array*) – The indicator what can be altered (0) and what can not (1)
- **dst** (*int*) – Parameter controlling behavior near edges, value could be probably deduced from the mask.

imreg_dft.utils.**getCuts**(*shp0*, *shp1*, *coef=0.5*)
Given an array shape, tile shape and density coefficient, return list of possible points of the array decomposition.

**Parameters**

- **shp0** (*np.ndarray*) – Shape of the big array
- **shp1** (*np.ndarray*) – Shape of the tile
- **coef** (*float*) – Density coefficient — lower means higher density and 1.0 means no overlap, 0.5 50% overlap, 0.1 90% overlap etc.

**Returns** List of tuples (y, x) coordinates of possible tile corners.

**Return type** list

imreg_dft.utils.**getSlices**(*inshp*, *outshp*, *coef*)

imreg_dft.utils.**get_apofield**(*shape*, *aporad*)
Returns an array between 0 and 1 that goes to zero close to the edges.

imreg_dft.utils.**get_best_cluster**(*points*, *scores*, *rad=0*)
Given some additional data, choose the best cluster and the index of the best point in the best cluster. Score of a cluster is sum of scores of points in it.

Note that the point of the best score may not be in the best cluster and a point may be members of multiple cluster.

**Parameters**

- **points** – Array of bools, indices that belong to the cluster are True
- **scores** – Rates a point by a number — higher is better.

imreg_dft.utils.**get_borderval**(*img*, *radius=None*)
Given an image and a radius, examine the average value of the image at most radius pixels from the edge

imreg_dft.utils.**get_clusters**(*points*, *rad=0*)
Given set of points and radius upper bound, return a binary matrix telling whether a given point is close to other points according to *_get_dst1()*. (point = matrix row).

**Parameters**

- **points** (*np.ndarray*) – Shifts.
- **rad** (*float*) – What is closer than `rad` is considered *close*.

The result matrix has always True on diagonals.

imreg_dft.utils.**get_values**(*cluster*, *shifts*, *scores*, *angles*, *scales*)
Given a cluster and some vectors, return average values of the data in the cluster. Treat the angular data carefully.

imreg_dft.utils.**imfilter**(*img*, *low=None*, *high=None*, *cap=None*)
Given an image, it a high-pass and/or low-pass filters on its Fourier spectrum.

**Parameters**

- **img** (*ndarray*) – The image to be filtered
- **low** (*tuple*) – The low-pass filter parameters, 0..1
- **high** (*tuple*) – The high-pass filter parameters, 0..1
- **cap** (*tuple*) – The quantile cap parameters, 0..1. A filtered image will have extremes below the lower quantile and above the upper one cut.

**Returns** The real component of the image after filtering

**Return type** np.ndarray

imreg_dft.utils.**mkCut**(*shp0*, *dims*, *start*)
Make a cut from shp0 and keep the given dimensions. Also obey the start, but if it is not possible, shift it backwards

**Returns** list - List of slices defining the subarray.

imreg_dft.utils.**rot180**(*arr*)
Rotate the input array over 180°

imreg_dft.utils.**slices2start**(*slices*)
Convenience function. Given a tuple of slices, it returns an array of their starts.

imreg_dft.utils.**starts2dshape**(*starts*)
Given starts of tiles, deduce the shape of the decomposition from them.

**Parameters** **starts** (*list of ints*) –

**Returns** shape of the decomposition

**Return type** tuple

imreg_dft.utils.**undo_embed**(*what*, *orig_shape*)
Undo an embed operation

**Parameters**

- **what** – What has once be the destination array
- **orig_shape** – The shape of the once original array

**Returns** The closest we got to the undo

imreg_dft.utils.**unextend_by**(*what*, *dst*)
Try to undo as much as the *extend_by()* does. Some things can't be undone, though.

imreg_dft.utils.**wrap_angle**(*angles*, *ceil=6.283185307179586*)

**Parameters**

- **angles** (float or ndarray, unit depends on kwarg ceil) –
- **ceil** (*float*) – Turnaround value

### tiles module

This module contains generic functionality for phase correlation, so it can be reused easily.

imreg_dft.tiles.**_assemble_resdict**(*ii*)

imreg_dft.tiles.**_distribute_resdict**(*resdict*, *ii*)

imreg_dft.tiles.**_fill_globals**(*tiles*, *poss*, *image*, *opts*)

imreg_dft.tiles.**_postprocess_unextend**(*ims*, *im2*, *extend*, *rcoef=1*)

imreg_dft.tiles.**_preprocess_extend**(*ims*, *extend*, *low*, *high*, *cut*, *rcoef*)

imreg_dft.tiles.**_preprocess_extend_single**(*im*, *extend*, *low*, *high*, *cut*, *rcoef*, *bigshape*)

imreg_dft.tiles.**filter_images**(*imgs*, *low*, *high*, *cut*)

imreg_dft.tiles.**process_images**(*ims*, *opts*, *tosa=None*, *get_unextended=False*, *reports=None*)

**Parameters**

- **tosa** (*np.ndarray*) – An array where to save the transformed subject.
- **get_unextended** (*bool*) – Whether to get the transformed subject in the same shape and coord origin as the template.

imreg_dft.tiles.**process_tile**(*ii*, *reports=None*)

imreg_dft.tiles.**resample**(*img*, *coef*)

imreg_dft.tiles.**settle_tiles**(*imgs*, *tiledim*, *opts*, *reports=None*)

### imreg module

This module contains mostly high-level functions.

FFT based image registration. — main functions

imreg_dft.imreg.**_get_ang_scale**(*ims*, *bgval*, *exponent='inf'*, *constraints=None*, *reports=None*)

Given two images, return their scale and angle difference.

**Parameters**

- **ims** (*2-tuple-like of 2D ndarrays*) – The images
- **bgval** – We also pad here in the `map_coordinates()`
- **exponent** (*float or 'inf'*) – The exponent stuff, see *similarity()*
- **constraints** (*dict, optional*) –
- **reports** (*optional*) –

**Returns** Scale, angle. Describes the relationship of the subject image to the first one.

**Return type** tuple

imreg_dft.imreg.**_get_log_base**(*shape*, *new_r*)

Basically common functionality of *_logpolar()* and *_get_ang_scale()*

This value can be considered fixed, if you want to mess with the logpolar transform, mess with the shape.

**Parameters**

- **shape** – Shape of the original image.
- **new_r** (*float*) – The r-size of the log-polar transform array dimension.

**Returns** Base of the log-polar transform. The following holds: $log\_base = \exp(\ln[spectrum\_dim]/loglpolar\_scale\_dim)$, or the equivalent $log\_base^{loglpolar\_scale\_dim} = spectrum\_dim$.

**Return type** float

imreg_dft.imreg.**_get_odds**(*angle*, *target*, *stdev*)

Determine whether we are more likely to choose the angle, or angle + 180°

**Parameters**

- **angle** (*float, degrees*) – The base angle.

- **target** (*float, degrees*) – The angle we think is the right one. Typically, we take this from constraints.
- **stdev** (*float, degrees*) – The relevance of the target value. Also typically taken from constraints.

**Returns**

**The greater the odds are, the higher is the preferrence** of the angle + 180 over the original angle. Odds of -1 are the same as inifinity.

**Return type** float

imreg_dft.imreg.**_get_pcorr_shape**(*shape*)

imreg_dft.imreg.**_get_precision**(*shape*, *scale=1*)

Given the parameters of the log-polar transform, get width of the interval where the correct values are.

**Parameters**

- **shape** (*tuple*) – Shape of images
- **scale** (*float*) – The scale difference (precision varies)

imreg_dft.imreg.**_logpolar**(*image*, *shape*, *log_base*, *bgval=None*)

Return log-polar transformed image Takes into account anisotropicity of the freq spectrum of rectangular images

**Parameters**

- **image** – The image to be transformed
- **shape** – Shape of the transformed image
- **log_base** – Parameter of the transformation, get it via *_get_log_base()*
- **bgval** – The backround value. If None, use minimum of the image.

**Returns** The transformed image

imreg_dft.imreg.**_logpolar_filter**(*shape*)

Make a radial cosine filter for the logpolar transform. This filter suppresses low frequencies and completely removes the zero freq.

imreg_dft.imreg.**_phase_correlation**(*im0*, *im1*, *callback=None*, **args*)

Computes phase correlation between im0 and im1

**Parameters**

- **im0** –
- **im1** –
- **callback** (*function*) – Process the cross-power spectrum (i.e. choose coordinates of the best element, usually of the highest one). Defaults to `imreg_dft.utils.argmax2D()`

**Returns**

**The translation vector (Y, X). Translation vector of (0, 0)** means that the two images match.

**Return type** tuple

imreg_dft.imreg.**_similarity**(*im0*, *im1*, *numiter=1*, *order=3*, *constraints=None*, *filter_pcorr=0*, *exponent='inf'*, *bgval=None*, *reports=None*)

This function takes some input and returns mutual rotation, scale and translation. It does these things during the process:

- Handles correct constraints handling (defaults etc.).
- Performs angle-scale determination iteratively. This involves keeping constraints in sync.

- Performs translation determination.
- Calculates precision.

**Returns** Dictionary with results.

imreg_dft.imreg.**_translation**(*im0*, *im1*, *filter_pcorr=0*, *constraints=None*, *reports=None*)
The plain wrapper for translation phase correlation, no big deal.

imreg_dft.imreg.**imshow**(*im0*, *im1*, *im2*, *cmap=None*, *fig=None*, ***kwargs*)
Plot images using matplotlib. Opens a new figure with four subplots:


**Parameters**

- **im0** (*np.ndarray*) – The template image
- **im1** (*np.ndarray*) – The subject image
- **im2** – The transformed subject — it is supposed to match the template
- **cmap** (*optional*) – colormap
- **fig** (*optional*) – The figure you would like to have this plotted on

**Returns** The figure with subplots

**Return type** matplotlib figure

imreg_dft.imreg.**similarity**(*im0*, *im1*, *numiter=1*, *order=3*, *constraints=None*, *filter_pcorr=0*, *exponent='inf'*, *reports=None*)
Return similarity transformed image im1 and transformation parameters. Transformation parameters are: isotropic scale factor, rotation angle (in degrees), and translation vector.

A similarity transformation is an affine transformation with isotropic scale and without shear.

**Parameters**

- **im0** (*2D numpy array*) – The first (template) image
- **im1** (*2D numpy array*) – The second (subject) image
- **numiter** (*int*) – How many times to iterate when determining scale and rotation
- **order** (*int*) – Order of approximation (when doing transformations). 1 = linear, 3 = cubic etc.
- **filter_pcorr** (*int*) – Radius of a spectrum filter for translation detection
- **exponent** (*float or 'inf'*) – The exponent value used during processing. Refer to the docs for a thorough explanation. Generally, pass "inf" when feeling conservative. Otherwise, experiment, values below 5 are not even supposed to work.
- **constraints** (*dict or None*) – Specify preference of seeked values. Pass None (default) for no constraints, otherwise pass a dict with keys `angle`, `scale`, `tx` and/or `ty` (i.e. you can pass all, some of them or none of them, all is fine). The value of a key is supposed to be a mutable 2-tuple (e.g. a list), where the first value is related to the constraint center and the second one to softness of the constraint (the higher is the number, the more soft a constraint is).

More specifically, constraints may be regarded as weights in form of a shifted Gaussian curve. However, for precise meaning of keys and values, see the documentation section *Using constraints*. Names of dictionary keys map to names of command-line arguments.

**Returns** Contains following keys: `scale`, `angle`, `tvec` (Y, X), `success` and `timg` (the transformed subject image)

**Return type** dict

---

**Note:** There are limitations

- Scale change must be less than 2.
- No subpixel precision (but you can use *resampling* to get around this).

---

imreg_dft.imreg.**similarity_matrix**(*scale*, *angle*, *vector*)

Return homogeneous transformation matrix from similarity parameters.

Transformation parameters are: isotropic scale factor, rotation angle (in degrees), and translation vector (of size 2).

The order of transformations is: scale, rotate, translate.

imreg_dft.imreg.**transform_img**(*img*, *scale=1.0*, *angle=0.0*, *tvec=(0, 0)*, *mode='constant'*, *bgval=None*, *order=1*)

Return translation vector to register images.

**Parameters**

- **img** (*2D or 3D numpy array*) – What will be transformed. If a 3D array is passed, it is treated in a manner in which RGB images are supposed to be handled - i.e. assume that coordinates are (Y, X, channels). Complex images are handled in a way that treats separately the real and imaginary parts.
- **scale** (*float*) – The scale factor (scale > 1.0 means zooming in)
- **angle** (*float*) – Degrees of rotation (clock-wise)
- **tvec** (*2-tuple*) – Pixel translation vector, Y and X component.
- **mode** (*string*) – The transformation mode (refer to e.g. `scipy.ndimage.shift()` and its kwarg `mode`).
- **bgval** (*float*) – Shade of the background (filling during transformations) If None is passed, *imreg_dft.utils.get_borderval()* with radius of 5 is used to get it.
- **order** (*int*) – Order of approximation (when doing transformations). 1 = linear, 3 = cubic etc. Linear works surprisingly well.

**Returns** The transformed img, may have another i.e. (bigger) shape than the source.

**Return type** np.ndarray

imreg_dft.imreg.**transform_img_dict**(*img*, *tdict*, *bgval=None*, *order=1*, *invert=False*)

Wrapper of *transform_img()*, works well with the *similarity()* output.

**Parameters**

- **img** –
- **tdict** (*dictionary*) – Transformation dictionary — supposed to contain keys "scale", "angle" and "tvec"
- **bgval** –
- **order** –
- **invert** (*bool*) – Whether to perform inverse transformation — doesn't work very well with the translation.

**Returns**

**See also:**

*transform_img()*

**Return type** np.ndarray

imreg_dft.imreg.**translation**(*im0*, *im1*, *filter_pcorr=0*, *odds=1*, *constraints=None*, *reports=None*)

Return translation vector to register images. It tells how to translate the im1 to get im0.

**Parameters**

- **im0** (*2D numpy array*) – The first (template) image
- **im1** (*2D numpy array*) – The second (subject) image
- **filter_pcorr** (*int*) – Radius of the minimum spectrum filter for translation detection, use the filter when detection fails. Values > 3 are likely not useful.
- **constraints** (*dict or None*) – Specify preference of seeked values. For more detailed documentation, refer to *similarity()*. The only difference is that here, only keys `tx` and/or `ty` (i.e. both or any of them or none of them) are used.
- **odds** (*float*) – The greater the odds are, the higher is the preferrence of the angle + 180 over the original angle. Odds of -1 are the same as inifinity. The value 1 is neutral, the converse of 2 is 1 / 2 etc.

**Returns**

**Contains following keys: angle, tvec (Y, X),** and `success`.

**Return type** dict

### 2.7.2 How to release

The build process in Python is not straightforward (as of 2014). Generally, you want this to be taken care of:

- The version mentioned in `src/imreg_dft/__init__.py` is the right one.
- Documentation can be generated (after make clean) and tests run OK too.
- The source tree is tagged (this is obviously the last step).

For this, there is a `bash` script `tests/release.sh`. It accepts one argument — the version string. It runs everything and although it doesn't do anything, it helps you to keep track of what is OK and what still needs to be worked on.

You can execute it from anywhere, for example from the project root:

```
[user@linuxbox imreg_dft]$ bash tests/release.sh 1.0.5
```

The output should be self-explanatory. The script is not supposed to rewrite anything important; however, it may run the documentation generation and tests. Those, however, can.

### 2.7.3 Become part of it!

Do you like the project? Do you feel inspired? Do you want to help out?

You are warmly welcome to do so!

### How to contribute

The process is pretty standard if you are used to Github.

most likely

1. Become familiar with git and learn how to use it properly, i.e. tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. You can do two things now:
   (a) Fork `imreg_dft` using Github web interface and clone it.
   (b) If you want to make a minor modification and/or don't have a Github account, just clone `imreg_dft`:

```
git clone https://github.com/matejak/imreg_dft
cd imreg_dft
```

3. Make a 'feature branch'. This will be where you work on your bug fix or whatever. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

   Then, do some edits, and commit them as you go.

4. Finally, you have to deliver your precious work in a smart way to the project. How to do this depends on whether you have created a pull request using Github or whether you went the simpler, but hardcore way. So, you have to do either
   (a) use again the Github interface, select your feature branch there and do some clicking stuff to create a pull request,
   (b) or make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

   You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

   Send these files to the current project maintainer.

---

**Note:** If you hack the code, remember these things:

- Add yourself into the `AUTHORS` file and briefly describe your contribution.
- If your contribution affects how `imreg_dft` works (this is *very* likely), mention this in the documentation.

---

## 2.8 User-centric changelog

#### 2.0.0 — 2016-06-19

- Preliminary support for sub-pixel transformation detection (watch #10).
- Changed interface for the `imreg_dft.imreg.translation` function (it returns a dict).

- New script: `ird-tform` transforms an image if given a transformation (closes #19).
- New script: `ird-show` allows in-depth visual insight into a phase correlation operation (closes #20).

**1.0.5 — 2015-05-02**

- Fixed project documentation typos, added the `AUTHORS` file.
- Added support for `pyfftw` for increased performance.
- Improved integration with MS Windows.
- Fixed an install bug (closes #18) that occured when dependencies were not met at install-time.
- Added documentation for Python constraint interface (closes #15).

**1.0.4 — 2015-03-03**

- Increased robustness of the tiling algorithm (i.e. when matching small subjects against large templates).
- Improved regression tests.
- Fixed project description typo.

**1.0.3 — 2015-02-22**

- Fixed the `MANIFEST.in` so the package is finally `easy_install`-able.
- Added the release check script stub.
- Updated install docs.

**1.0.2 — 2015-02-21**

- Documentation and setup.py fixes.

**1.0.1 — 2015-02-19**

- Real debut on PyPi.
- Fixed some minor issues with setup.py and docs.

**1.0.0 — 2015-02-19**

Beginning of the changelog.

- Debut on PyPi

CHAPTER 3

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## i

imreg_dft.imreg, 24
imreg_dft.loader, 18
imreg_dft.tiles, 23
imreg_dft.utils, 19

# Index

## Symbols


_ang2complex() (in module imreg_dft.utils), 19
_apodize() (in module imreg_dft.utils), 19
_argmax2D() (in module imreg_dft.utils), 19
_argmax_ext() (in module imreg_dft.utils), 19
_assemble_resdict() (in module imreg_dft.tiles), 23
_calc_tform() (in module imreg_dft.utils), 20
_calc_tform_complete() (in module imreg_dft.utils), 20
_compensate_fftshift() (in module imreg_dft.utils), 20
_complex2ang() (in module imreg_dft.utils), 20
_distribute_resdict() (in module imreg_dft.tiles), 23
_extend_array() (in module imreg_dft.utils), 20
_fill_globals() (in module imreg_dft.tiles), 23
_getCut() (in module imreg_dft.utils), 20
_get_ang_scale() (in module imreg_dft.imreg), 24
_get_angles() (in module imreg_dft.utils), 20
_get_constraint_mask() (in module imreg_dft.utils), 20
_get_dst1() (in module imreg_dft.utils), 20
_get_emslices() (in module imreg_dft.utils), 20
_get_log_base() (in module imreg_dft.imreg), 24
_get_lograd() (in module imreg_dft.utils), 20
_get_odds() (in module imreg_dft.imreg), 24
_get_pcorr_shape() (in module imreg_dft.imreg), 25
_get_precision() (in module imreg_dft.imreg), 25
_get_subarr() (in module imreg_dft.utils), 20
_get_success() (in module imreg_dft.utils), 20
_highpass() (in module imreg_dft.utils), 21
_interpolate() (in module imreg_dft.utils), 21
_load2reg() (imreg_dft.loader.Loader method), 18
_logpolar() (in module imreg_dft.imreg), 25
_logpolar_filter() (in module imreg_dft.imreg), 25
_lowpass() (in module imreg_dft.utils), 21
_phase_correlation() (in module imreg_dft.imreg), 25
_postprocess_unextend() (in module imreg_dft.tiles), 23
_preprocess_extend() (in module imreg_dft.tiles), 23
_preprocess_extend_single() (in module imreg_dft.tiles), 24
_save() (imreg_dft.loader.Loader method), 18
_similarity() (in module imreg_dft.imreg), 25
_translation() (in module imreg_dft.imreg), 26
_xpass() (in module imreg_dft.utils), 21


## A


argmax_angscale() (in module imreg_dft.utils), 21
argmax_translation() (in module imreg_dft.utils), 21


## D


decompose() (in module imreg_dft.utils), 21


## E


embed_to() (in module imreg_dft.utils), 21
extend_by() (in module imreg_dft.utils), 21
extend_to() (in module imreg_dft.utils), 21
extend_to_3D() (in module imreg_dft.utils), 21


## F


filter_images() (in module imreg_dft.tiles), 24
flatten() (in module imreg_dft.loader), 19
frame_img() (in module imreg_dft.utils), 22


## G


get_apofield() (in module imreg_dft.utils), 22
get_best_cluster() (in module imreg_dft.utils), 22
get_borderval() (in module imreg_dft.utils), 22
get_clusters() (in module imreg_dft.utils), 22
get_values() (in module imreg_dft.utils), 22
getCuts() (in module imreg_dft.utils), 22
getSlices() (in module imreg_dft.utils), 22
guessCanLoad() (imreg_dft.loader.Loader method), 18


## I


imfilter() (in module imreg_dft.utils), 22
imreg_dft.imreg (module), 24
imreg_dft.loader (module), 18
imreg_dft.tiles (module), 23
imreg_dft.utils (module), 19
imshow() (in module imreg_dft.imreg), 26


## L


load2reg() (imreg_dft.loader.Loader method), 18
Loader (class in imreg_dft.loader), 18
loader_of() (in module imreg_dft.loader), 19


## M


mkCut() (in module imreg_dft.utils), 23

## P

process_images() (in module imreg_dft.tiles), 24
process_tile() (in module imreg_dft.tiles), 24

## R

resample() (in module imreg_dft.tiles), 24
rot180() (in module imreg_dft.utils), 23

## S

save() (imreg_dft.loader.Loader method), 19
settle_loaders() (in module imreg_dft.loader), 19
settle_tiles() (in module imreg_dft.tiles), 24
similarity() (in module imreg_dft.imreg), 26
similarity_matrix() (in module imreg_dft.imreg), 27
slices2start() (in module imreg_dft.utils), 23
spawn() (imreg_dft.loader.Loader method), 19
starts2dshape() (in module imreg_dft.utils), 23

## T

transform_img() (in module imreg_dft.imreg), 27
transform_img_dict() (in module imreg_dft.imreg), 27
translation() (in module imreg_dft.imreg), 28

## U

undo_embed() (in module imreg_dft.utils), 23
unextend_by() (in module imreg_dft.utils), 23

## W

wrap_angle() (in module imreg_dft.utils), 23